GD2S03
Advanced Software Engineering & Programming for Games

MEDIA DESIGN SCHOOL

Bachelor of Software Engineering(BSE)
Game Development

- Overview
  - Generics
  - Automatic Reference Counting
  - Memory Safety
  - Access Control

Generics

- *Generic code* enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define

Generic Functions

- *Generic functions* can work with any type.
- The generic version of the function uses a *placeholder* type name (called T, in this case) instead of an *actual* type name (such as Int, String, or Double).

```swift
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

Type Parameters

- In the swapTwoValues(_:_:) example above, the placeholder type T is an example of a *type parameter*. Type parameters specify and name a placeholder type, and are written immediately after the function's name, between a pair of matching angle brackets (such as <T>).

- The type parameter is replaced with an *actual* type whenever the function is called.

Generic - Types

- In addition to generic functions, Swift enables you to define your own *generic types*. These are custom classes, structures, and enumerations that can work with *any* type, in a similar way to **Array** and **Dictionary**.

- In most cases, type parameters have descriptive names, such as **Key** and **Value** in **Dictionary<Key, Value>**and **Element** in **Array<Element>**

- However, when there isn't a meaningful relationship between them, it's traditional to name them using single letters such as **T**, **U**, and **V**,

```swift
struct Stack<Element> {
  var items = [Element]()
  mutating func push(_ item: Element) {
    items.append(item)
  }
  mutating func pop() -> Element {
    return items.removeLast()
  }
}
```

```swift
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings

let fromTheTop = stackOfStrings.pop()
// fromTheTop is equal to "cuatro", and the
stack now contains 3 strings
```

Generics

Extending Generics Types

- When a generic type is extended the type parameter list from the original list is available within the body of the extension.
- In this example, **Stack** type's existing type parameter name, **Element**, is used within the extension.

```
extension Stack {
  var topItem: Element? {
    return items.isEmpty ? nil :
      items[items.count - 1]
  }
}
if let topItem = stackOfStrings.topItem {
  print("The top item on the stack is \(topItem).")
}// Prints "The top item on the stack is tres."
```

Type Constraints

- Enforcing certain *type constraints* specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.
- The single type parameter for findIndex(of:in:) is written as T: Equatable, which means "any type T that conforms to the Equatable protocol."

```
func findIndex<T: Equatable>(of valueToFind: T, in
array:[T]) -> Int? {
  for (index, value) in array.enumerated() {
    if value == valueToFind {
      return index
    }
  }
  return nil
}
let doubleIndex = findIndex(of: 9.3, in: [3.14159, 0.1, 0.25])
//  9.3 isn't in the array
let stringIndex = findIndex(of: "Andrea", in: ["Mike", "Malcolm", "Andrea"])
// stringIndex containing a value of 2
```

Generics – Associated Types

- Associated types are type placeholders in Protocol definition whose actual type is decided when it's adopted.
- Associated types are specified with the `associatedtype` keyword.

```
protocol Container {
  associatedtype Item
  mutating func append(_ item: Item)
  var count: Int { get }
  subscript(i: Int) -> Item { get }
}
```

```
struct Stack<Element>: Container {
    // original Stack<Element> implementation
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
 // conformance to the Container protocol
    mutating func append(_ item: Element) {
        self.push(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Element {
        return items[i]
    }}
```

- The type parameter ***Element*** is used as the type of the append(_:) method's item parameter and the return type of the subscript. Swift can therefore infer that ***Element*** is the appropriate type to use as the Item for this particular container.

Generics – Associated Types

Extending an Existing Type to Specify an Associated Type

- An existing type which can be extended to add conformance to a protocol.
- Array type already provides append(_:) method, count property and a subscript with int index to retrieve its' element hence it can be extended to conform the container protocol

```
extension Array: Container {}
```

Using Type Annotations to Constrain an Associated Type

- Type annotation can be added to an associated type in a protocol, to require that conforming types satisfy the constraints described by the type annotation.
- To conform to this version of Container, the container's Item type has to conform to the Equatable protocol

```
protocol Container {
  associatedtype Item: Equatable
  mutating func append(_ item: Item)
  var count: Int { get }
  subscript(i: Int) -> Item { get }
}
```

Using a Protocol in Its Associated Type's Constraints

- In this protocol, Suffix is an associated type, like the Item type in the Container example above. Suffix has two constraints: It must conform to the **SuffixableContainer** protocol (the protocol currently being defined), and its Item type must be the same as the container's Item type

```
protocol SuffixableContainer: Container {
 associatedtype Suffix: SuffixableContainer
 where Suffix.Item == Item
   func suffix(_ size: Int) -> Suffix
}
```

Generic

Generic Where Clause

- A requirement for associated type is defined by a *generic where clause*.
- A generic **where** clause starts with the **where** keyword, followed by constraints for associated types or equality relationships between types and associated types.
- You write a generic **where** clause right before the opening curly brace of a type or function's body.

```
func allItemsMatch<C1: Container, C2: Container>
  (_ someContainer: C1, _ anotherContainer: C2) -> Bool
  where C1.Item == C2.Item, C1.Item: Equatable {
// Both containers contain the same number of items.
    if someContainer.count != anotherContainer.count {
      return false }
// Check each pair of items to see if they're equivalent.
    for i in 0..<someContainer.count {
      if someContainer[i] != anotherContainer[i] {
        return false}
    }
// All items match, so return true.
    return true
}
```

Extensions with a Generic Where Clause

- You can also use a generic where clause as part of an extension.
- The example extends the generic Stack structure from the previous examples to add an isTop(_:) method.

```
extension Stack where Element: Equatable {
  func isTop(_ item: Element) -> Bool {
    guard let topItem = items.last else {
      return false
    }
    return topItem == item
  }
}
```

Generics

Associated Types with a Generic Where Clause

- You can include a generic **where** clause on an associated type.
- The generic **where** clause on **Iterator** requires that the iterator must traverse over elements of the same item type as the container's items, regardless of the iterator's type

```
protocol Container {
  associatedtype Item
  mutating func append(_ item: Item)
  var count: Int { get }
  subscript(i: Int) -> Item { get }
  associatedtype Iterator: IteratorProtocol where
     Iterator.Element == Item
  func makeIterator() -> Iterator
}
```

Generic Subscripts

- Subscripts can be generic, and they can include generic where clauses.
- You write the placeholder type name inside angle brackets after subscript, and you write a generic where clause right before the opening curly brace of the subscript's body.

```
extension Container {
subscript<Indices: Sequence>(indices: Indices)->[Item]
  where Indices.Iterator.Element == Int {
    var result = [Item]()
    for index in indices {
      result.append(self[index])
    }
    return result
  }
}
```

Automatic Reference Counting (ARC)

- Every time a new instance of a class is created, ARC allocates a chunk of memory to store information about that instance.
- This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.
- Additionally, when an instance is no longer needed, ARC frees up the memory.
- However, ARC will not deallocate an instance as long as at least one active reference to that instance still exists.
- To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a strong reference to the instance.

```
class Person {
  let name: String
  init(name: String) {
    self.name = name
    print("\(name) is being initialized")
  }
  deinit {
    print("\(name) is being deinitialized")
  }
}
var reference1: Person?
var reference2: Person?
var reference3: Person?

reference1 = Person(name: "John Appleseed")
// Prints "John Appleseed is being initialized"
reference2 = reference1
reference3 = reference1

reference1 = nil
reference2 = nil
reference3 = nil
// Prints "John Appleseed is being deinitialized"
```

Strong Reference Cycle

- A *strong reference cycle* can be created when two class instances hold a strong reference to each other, such that each instance keeps the other alive.
- **john** and **unit4a** variables have strong reference to Person and Apartment instance respectively.
- Now two instances can be linked together so the person has an apartment and the apartment has a tenant again having strong reference.
- Assigning nil value to john and unit4A variable does not call the deinitializer.

```
class Person {
  let name: String
  init(name: String) {
    self.name = name
  }
  var apartment: Apartment?
  deinit {
     print("\(name) is
     being deinitialized")
  }
}

class Apartment {
  let unit: String
  init(unit: String) {
    self.unit = unit
  }
  var tenant: Person?
  deinit {
     print("Apartment \(unit)
      is being deinitialized")
  }
}
```

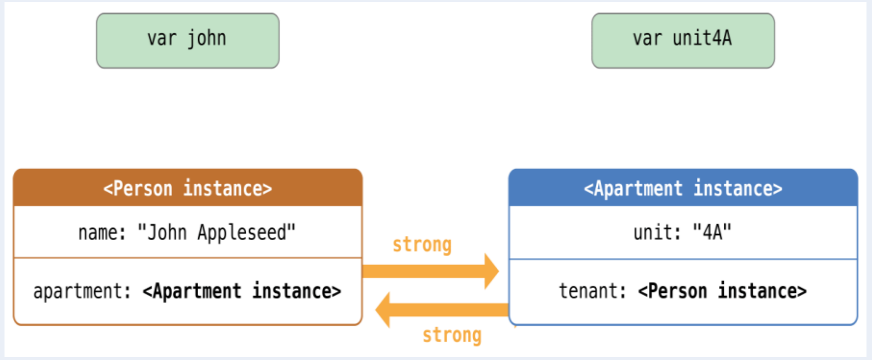
```
var john: Person?
var unit4A: Apartment?

John = Person( name:
               "John Appleseed")

unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john

john = nil
unit4A = nil
```

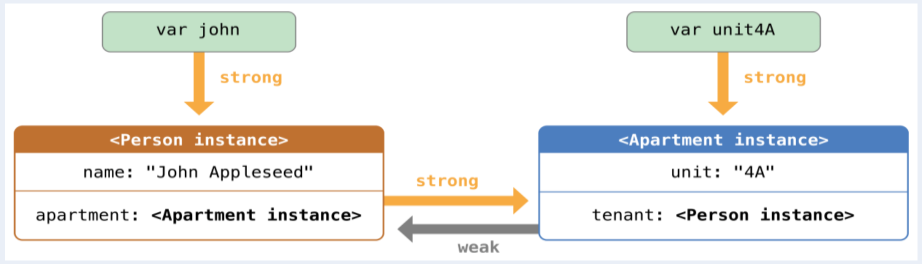


Resolving Strong Reference Cycles Between Class Instance - weak

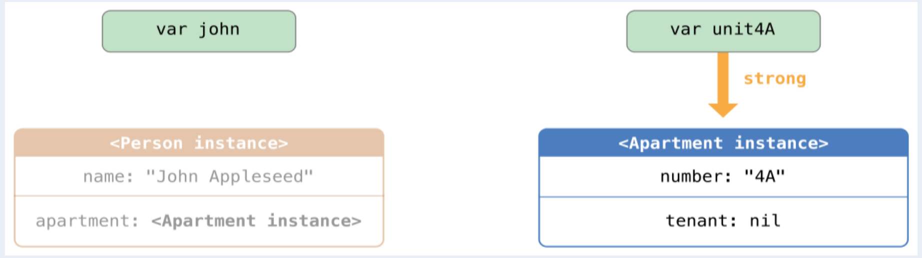
- Swift provides two ways to resolve strong reference cycles when you work with properties of class type: **weak references and unowned references.**
- Use a weak reference when the other instance has a shorter lifetime—that is, when the other instance can be deallocated first.
- It is appropriate for an apartment not to have no tenant and hence it's tenant variable can be weak.
- In the previous example replace line

```
var tenant: Person?
      with
weak var tenant: Person?
```

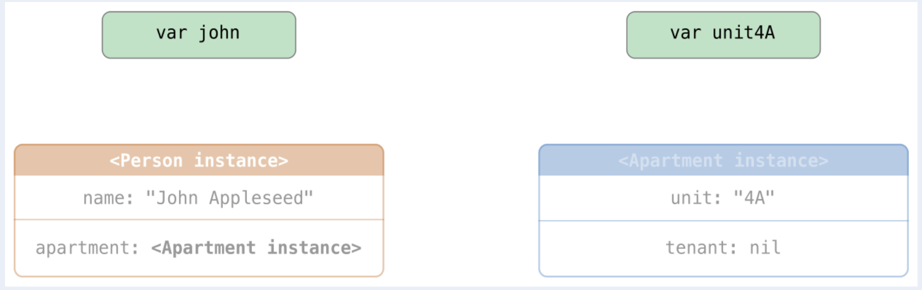
- The Person instance still has a ***strong*** reference to the Apartment instance, but the Apartment instance now has a ***weak*** reference to the Person instance.



```
john = nil
 //Prints "John Appleseed is being deinitialized"
```



```
unit4A = nil
 // Prints "Apartment 4A is being deinitialized"
```

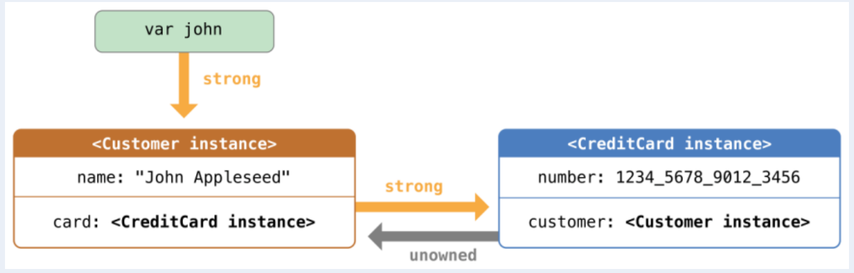


Resolving Strong Reference Cycles Between Class Instance - Unowened

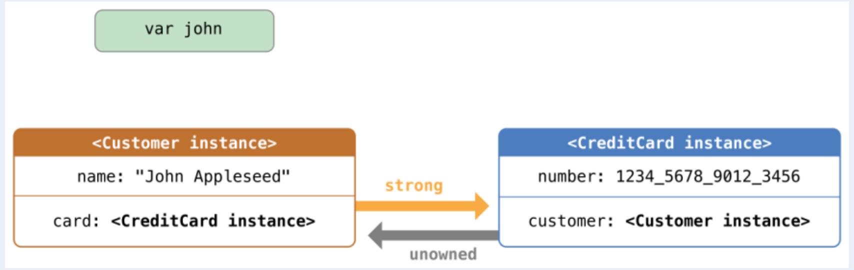
- *unowned reference* does not keep a strong hold on the instance it refers to.
- An unowned reference is used when the other instance has the same lifetime or a longer lifetime.
- ***unowned*** keyword is placed before a property or variable declaration.
- An unowned reference is expected to always have a value. As a result, ARC never sets an unowned reference's value to nil, which means that unowned references are defined using non-optional types.

```
class Customer {
  let name: String
  var card: CreditCard?
  init(name: String) {
    self.name = name
  }
  deinit { print("\(name) is being
    deinitialized") }
}
class CreditCard {
  let number: UInt64
  unowned let customer: Customer
  init(number: UInt64,
      customer:  Customer) {
    self.number = number
    self.customer = customer
  }
  deinit { print("Card #\(number) is
being deinitialized") }
}
```

```
var john: Customer?
john = Customer(name: "John
Appleseed")
john!.card = CreditCard(number:
1234_5678_9012_3456, customer:
john!)
```



```
john = nil
// Prints "John Appleseed is being
deinitialized"
// Prints "Card #1234567890123456
is being deinitialized"
```


var john
<Customer instance>
name: "John Appleseed"
card: <CreditCard instance>
strong
unowned
<CreditCard instance>
number: 1234_5678_9012_3456
customer: <Customer instance>

Unowned References and Implicitly Unwrapped Optional Properties

- Now consider a scenario, in which both properties should always have a value, and neither property should ever be **nil** once initialization is complete.
- In this scenario, it’s useful to combine an unowned property on one class with an implicitly unwrapped optional property on the other class.

```
class Country {
  let name: String
  var capitalCity: City!
  init(name: String, capitalName: String) {
    self.name = name
    self.capitalCity = City(name: capitalName,
                            country: self)
  }
}
class City {
  let name: String
  unowned let country: Country
  init(name: String, country: Country) {
    self.name = name
    self.country = country
  }
}
```

```
var country = Country(name: "Canada",
                      capitalName: "Ottawa")
print("\(country.name)'s capital city is called
\(country.capitalCity.name)")
// Prints "Canada's capital city is called Ottawa"
```

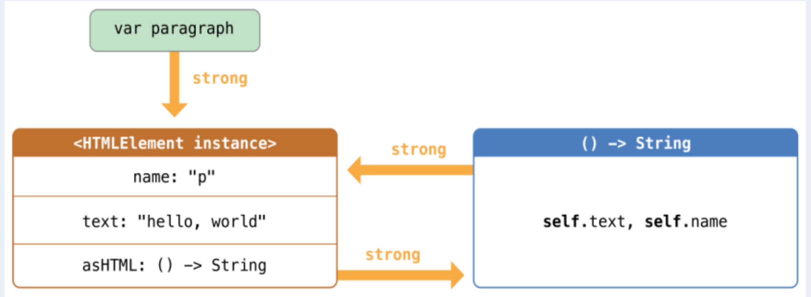
- In the example above, the use of an implicitly unwrapped optional means that all of the two-phase class initializer requirements are satisfied.
- The capitalCity property can be used and accessed like a nonoptional value once initialization is complete, while still avoiding a strong reference cycle

Strong Reference Cycle for Closures

- A strong reference cycle can also occur if you assign a closure to a property of a class instance, and the body of that closure captures the instance.
- This capture might occur because the closure's body accesses a property of the instance, such as **self.someProperty**, or because the closure calls a method on the instance, such as **self.someMethod()**
- In either case, these accesses cause the closure to "capture" **self**, creating a strong reference cycle

```
class HTMLElement {
  let name: String
  let text: String?
  lazy var asHTML: () -> String = {
    if let text = self.text {
      return
"<\(self.name)>\(text)</\(self.name
)>"
    } else {
      return "<\(self.name) />"
    }
  }
  init(name: String,
     text: String? = nil) {
      self.name = name
      self.text = text
    }
  deinit { print("\(name) is
    being deinitialized")
  }
}
```

```
var paragraph: HTMLElement? =
HTMLElement(name: "p",
          text: "hello,world")
print(paragraph!.asHTML())
// Prints "<p>hello,world</p>"
```



- var when set to nil causes strong reference cycle.
- This strong reference cycle occurs because closures, like classes, are *reference types*

Resolving Strong Reference Cycles for Closures

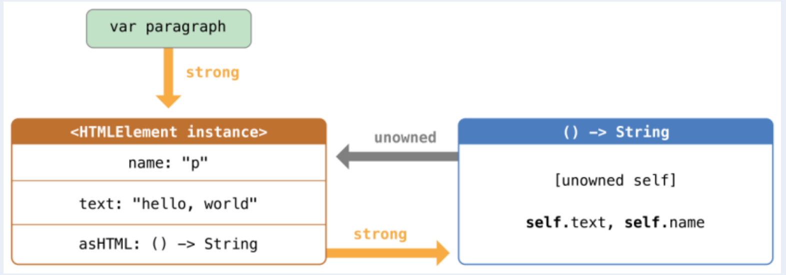
- Define a *capture list* is as part of the closure's definition.
- Each captured reference is declared to be a weak or unowned reference rather than a strong reference.
- Define a capture as a weak reference when the captured reference may become **nil** at some point in the future.
- Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and **will always be deallocated at the same time.**

- Each item in a capture list is a pairing of the **weak or unowned** keyword with a reference to a class instance (such as **self**) or a variable initialized with some value (such as **delegate = self.delegate!**).
- These pairings are written within a pair of square braces, separated by commas.

```
lazy var someClosure: (Int,
String) -> String = {
[unowned self, weak delegate =
self.delegate!] (index: Int,
stringToProcess: String) -> String
in
// closure body goes here
}
```

In the previous define the closure with capture list as

```
lazy var asHTML:()->String = {
  [unowned self] in
  if let text = self.text {
    return "<\(self.name)>
    \(text)</\(self.name)>"
  } else {
    return "<\(self.name) />"
  }
}
```



Memory Safety

- A conflicting access to memory can occur when different parts of your code are trying to access the same location in memory at the same time.
- Multiple accesses to a location in memory at the same time can produce unpredictable or inconsistent behavior.
- Specifically, a conflict occurs if two accesses meet all of the following conditions:
    - At least one is a write access.
    - They access the same location in memory.
    - Their durations overlap.
- An access is *instantaneous* if it's not possible for other code to run after that access starts but before it ends.
- An access is *long-term access*, if it's possible for other code to run after a long-term access starts but before it ends, which is called *overlap*.
- A long-term access can overlap with other long-term accesses and instantaneous accesses.

Memory Safety

Conflicting Access to In-Out Parameters

- A function has long term write access in-out to all of it’s in-out parameters.
- The write access for an in-out parameter lasts for the entire duration of that function call.
- One consequence of this long-term write access is that you can’t access the original variable that was passed as in-out

```
var stepSize = 1
  func incrementInPlace(_ number: inout Int) {
    number += stepSize
  }
incrementInPlace(&stepSize)
// Error: conflicting accesses to stepSize

// Make an explicit copy.
var copyOfStepSize = stepSize
incrementInPlace(&copyOfStepSize)
```

- Another consequence of long-term write access to in-out parameters is that passing a single variable as the argument for multiple in-out parameters of the same function produces a conflict. For example:

```
func balance(_ x: inout Int, _ y: inout Int) {
  let sum = x + y
  x = sum / 2
  y = sum - x
}

var playerOneScore = 42
var playerTwoScore = 30
balance(&playerOneScore, &playerTwoScore) // OK

balance(&playerOneScore, &playerOneScore)
// Error: Conflicting accesses to playerOneScore
```

Memory Safety – Conflicting Access to self in Methods

- A mutating method on a structure has write access to **self** for the duration of the method call.
- In the first call to sharehealth function with maria as an inout parameter there is no conflict as the memory location is different.
- In the second call to the function, the mutating method needs write access to **self** for the duration of the method, and the in-out parameter needs write access to **teammate** for the same duration.
- Within the method, both **self** and **teammate** refer to the same location in memory.
- The two write accesses refer to the same memory and they overlap, producing a conflict.

```
struct Player {
  var name: String
  var health: Int
  var energy: Int
  static let maxHealth = 10
  mutating func restoreHealth() {
    health = Player.maxHealth
  }
}

extension Player {
  mutating func shareHealth(with teammate: inout Player){
    balance(&teammate.health, &health)
  }
}

var oscar = Player(name: "Oscar", health: 10, energy: 10)
var maria = Player(name: "Maria", health: 5, energy: 10)
oscar.shareHealth(with: &maria) // OK

oscar.shareHealth(with: &oscar)
// Error: conflicting accesses to oscar
```

Memory Safety – Conflicting Access to Properties

- For enumeartions, structures or tuples, read write access to one of the properties requires read or write access to whole data.

- In this example there are two write accesses to holly with durations that overlap, hence causing a conflict.
- However swift prove that overlapping access to properties of a structure is safe if the following conditions apply:
  - only stored properties of an instance are accessed, not computed properties or class properties.
  - The structure is the value of a local variable, not a global variable.
  - The structure is either not captured by any closures, or it's captured only by nonescaping closures.
- If the compiler can't prove the access is safe, it doesn't allow the access.

```
var holly = Player(name: "Holly", health: 10, energy: 10)
balance(&holly.health, &holly.energy) // Error
```

```
func someFunction() {
  var oscar = Player(name: "Oscar", health: 10, energy: 10)
  balance(&oscar.health, &oscar.energy) // OK
}
```

Access Control – Access Levels

There are five types of access levels. By default it's internal	
`open`	• Open classes can be subclassed within the module where they're defined, and within any module that imports the module where they're defined. • Open class members can be overridden by subclasses within the module where they're defined, and within any module that imports the module where they're defined.
`public`	• Classes with public access, or any more restrictive access level, can be subclassed only within the module where they're defined. • Class members with public access, or any more restrictive access level, can be overridden by subclasses only within the module where they're defined.
`internal`	• Internal access enables entities to be used within any source file from their defining module, but not in any source file outside of that module.
`fileprivate`	• File-private access restricts the use of an entity to its own defining source file.
`private`	• Private access restricts the use of an entity to the enclosing declaration, and to extensions of that declaration that are in the same file

Access Control - Subclassing

- You can subclass any class that can be accessed in the current access context.
- A subclass can't have a higher access level than its superclass—for example, you can't write a public subclass of an internal superclass.
- You can override any class member (method, property, initializer, or subscript) that is visible in a certain access context.
- An override can make an inherited class member more accessible than its superclass version.
- It's even valid for a subclass member to call a superclass member that has lower access permissions than the subclass member, as long as the call to the superclass's member takes place within an allowed access level context

```
public class A {
    fileprivate func someMethod() {}
}

internal class B: A {
    override internal func someMethod() {
        super.someMethod()
    }
}
```

Access Control – Constants, Variables, Properties and Subscripts

- A constant, variable, or property can't be more public than its type.
- It's not valid to write a public property with a private type, for example.
- Similarly, a subscript can't be more public than either its index type or return type.
- If a constant, variable, property, or subscript makes use of a private type, the constant, variable, property, or subscript must also be marked as private:

```
private var privateInstance = SomePrivateClass()
```

- Getters and setters for constants, variables, properties, and subscripts automatically receive the same access level as the constant, variable, property, or subscript they belong to.
- A setter can be given a *lower* access level than its corresponding getter, to restrict the read-write scope.
- A lower access level is assigned by writing fileprivate(set), private(set), or internal(set) before the var or subscript introducer

```
struct TrackedString {
  private(set) var numberOfEdits = 0
  var value: String = "" {
    didSet {
      numberOfEdits += 1
    }
  }
}
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment
numberOfEdits."
stringToEdit.value += " So will this one."
print("The number of edits is
\(stringToEdit.numberOfEdits)")
// Prints "The number of edits is 3"
```

Access Control - Initializers

Initializers

- Custom initializers can be assigned an access level less than or equal to the type that they initialize.
- However, A required initializer must have the same access level as the class it belongs to.
- The types of an initializer's parameters can't be more private than the initializer's own access level.

Default Initializers

- A default initializer has the same access level as the type it initializes, unless that type is defined as public.
- For a type that is defined as public, the default initializer is considered internal
- If you want a public type to be initializable with a no-argument initializer when used in another module, you must explicitly provide a public no-argument initializer yourself as part of the type's definition.

Default memberwise Initializers for Structures Type

- The default memberwise initializer for a structure type is considered private if any of the structure's stored properties are private.
- Likewise, if any of the structure's stored properties are file private, the initializer is file private. Otherwise, the initializer has an access level of internal.

Access Control - Protocols

- Explicit access level is assigned to a protocol at the time of its definition.
- The access level of each requirement within a protocol definition is automatically set to the same access level as the protocol.
- If you define a new protocol that inherits from an existing protocol, the new protocol can have at most the same access level as the protocol it inherits from. You can’t write a public protocol that inherits from an *internal* protocol for example
- A type can conform to a protocol with a lower access level than the type itself.
- if a public type conforms to an internal protocol, the type’s implementation of each protocol requirement must be at least “internal”.

Access Control - Extensions

- You can extend a class, structure, or enumeration in any access context in which the class, structure, or enumeration is available.
- Any type members added in an extension have the same default access level as type members declared in the original type being extended.
- If you extend a public or internal type, any new type members you add have a default access level of internal.
- If you extend a file-private type, any new type members you add have a default access level of file private.
- If you extend a private type, any new type members you add have a default access level of private.

Extensions that are in the same file as the class, structure, or enumeration that they extend behave as if the code in the extension had been written as part of the original type’s declaration. As a result, you can:

- Declare a private member in the original declaration, and access that member from extensions in the same file.
- Declare a private member in one extension, and access that member from another extension in the same file.
- Declare a private member in an extension, and access that member from the original declaration in the same file.